Scalable black-box prediction models for multi-dimensional adaptation on NUMA multi-cores

Khasymski, A., & Nikolopoulos, D. S. (2015). Scalable black-box prediction models for multi-dimensional adaptation on NUMA multi-cores. *International Journal of Parallel, Emergent and Distributed Systems*, *30*(3), 193-210. https://doi.org/10.1080/17445760.2014.895346

Published in:
International Journal of Parallel, Emergent and Distributed Systems

Document Version:
Peer reviewed version

The International Journal of Parallel, Emergent and Distributed Systems
Vol. 00, No. 00, Month 200x, 1–18

RESEARCH ARTICLE

Scalable Black-Box Prediction Models for Multi-Dimensional Adaptation on NUMA Multi-Cores

Aleksandr Khasymski[a*] and Dimitrios S. Nikolopoulos[b]

[a]*Department of Computer Science, Virginia Tech, Blacksburg, VA 24060, USA*; [b]*School of EEECS, Queen's University of Belfast, Belfast BT7 1NN, United Kingdom*

(*v1.0 released March 2013*)

This paper presents a scalable, statistical "black-box" model for predicting the performance of parallel programs on multi-core NUMA systems. We derive a model with low overhead, by reducing data collection and model training time . The model can accurately predict the behavior of parallel applications in response to changes in their concurrency, thread layout on NUMA nodes, and core voltage and frequency. We present a framework that applies the model to achieve significant energy and energy-delay-square (ED^2) savings (9% and 25% respectively) along with performance improvement (10% mean) on an actual 16-core NUMA system running realistic application workloads. Our prediction model proves substantially more accurate than previous efforts.

Keywords:
black-box model; performance model; parallel applications; NUMA; energy; concurrency

1. Introduction

The dimensionality and complexity of performance analysis and optimization on parallel architectures keeps increasing. This happens at a faster rate than the rate at which research deciphers the performance implications of systems and software on parallel applications. On any given parallel architecture, an application developer is faced with a large number of tunable system parameters that interact in non-trivial and non-tractable ways to determine performance. The introduction of additional constraints in parallel execution, such as power and thermal constraints, complicate the task of performance optimization even further. These challenges make analytical models of system performance increasingly hard to construct and ultimately inaccurate, even in architecture-specific, application-specific or input-specific contexts [1–3]. These difficulties, along with the increasing diversity in parallel architectures, give rise to statistical, black-box techniques for performance prediction on parallel architectures [4–11].

Performance prediction using black-box models has several advantages. Users can exploit such predictors with little effort to select tunable parameters, such as the number of cores, processors and nodes used to run a parallel application [4, 11, 12], or application-specific input parameters [9]. Similar models can drive the selection of compiler optimizations [10] and runtime system parameter settings, including settings that optimize according to multiple criteria (such as performance, power and temperature [5]). Black-box models tend to yield performance predictions

*Corresponding author. Email: khasymskia@cs.vt.edu

ISSN: 1023-6198 print/ISSN 1563-5120 online

DOI: 10.1080/1023619YYxxxxxxxx
http://www.informaworld.com

rapidly on vast configuration spaces, using a relatively small number of execution samples. More importantly, black-box approaches are generally applicable across hardware platforms and applications.

This paper presents a new performance prediction model for parallel applications that strives to achieve five goals: (1) be a black-box model that does not require architectural insight or application-specific input from the user; (2) be scalable, both in terms of model training time, and in terms of applying the model with low overhead to rapidly predict performance; (3) predict performance in response to parameter changes of a large and multi-dimensional system configuration space; (4) be easy to implement on current architectures with existing hardware support; (5) address limitations and shortcomings of prior online models, in particular, performance sensitivity of parallel applications to the layout of their threads on cores and their data on memory banks.

The architectural testbed that we explore in this paper is a NUMA multiprocessor node with multi-core processors. Arguably, this is one of the most common commodity node architectures in HPC and data center settings, such as those featuring recent Intel processors based on QPI interconnects and AMD processors based on HyperTransport interconnects [13–16]. NUMA multi-cores, a seemingly well-understood and thoroughly studied architecture [17, 18], exposes several tunable parameters, including the number of processors, the number of cores per processor, the layout of active processors and active cores in each processor, and the layout of data in memory. If power is an additional optimization criterion, the voltage and frequency of each core needs to be tuned as well. Interestingly enough, NUMA systems tend to have asymmetries due to mismatch between thread placement and data placement [19, 20].

We present a model based on linear regression, with the following contributions:

- The model delivers excellent prediction accuracy – with error less than 7% on average.
- It effectively partitions the large number of possible execution configurations of a NUMA multi-processor system into a four-dimensional space.
- It is based on a single, portable metric, *per thread instructions per cycle*, that can be estimated along each dimension using simple linear regression, derived from samples at end points in the configuration space.
- Unlike other techniques, no lengthy training stage [5] or exhaustive search of the space is required, therefore the model drastically reduces overhead and improves portability, applicability, and ease of use.
- Prediction does not rely on any architecture-specific metrics that need to be identified by the user.

The rest of the paper is organized as follows. Section 2 provides background and preliminary concepts. Section 3 discusses the design of our multi-dimensional prediction model. We present our experimental analysis in Section 4. Section 5 discusses related work and Section 6 concludes the paper.

2. Preliminaries

We develop a performance model for iterative parallel HPC applications executed on NUMA shared memory systems with multiple multi-core processors. In this section we highlight how this choice affects the model and its derivation.

We exploit the iterative structure of HPC applications to reduce the sampling overhead of our model by observing that certain execution patterns, such as scalability and memory accesses, remain constant between iterations, either through

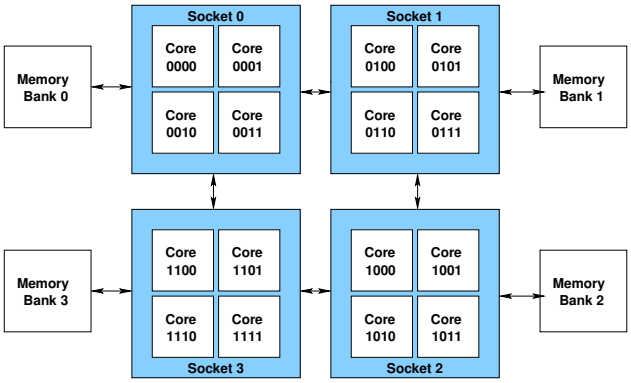

Figure 1. Four Node NUMA System. Sockets and cores are numbered in a cyclic manner according to the numbering of our AMD 16-core NUMA test system.

an entire run or through periods of a run. As such, we can infer, with considerable accuracy, the overall scalability of a given benchmark by running either a single iteration, or by periodically sampling iterations to adjust to irregular application behavior. In benchmarks with hundreds of iterations, the sampling overhead of the model is quickly amortized.

Our model is phase-aware, meaning that we subdivide each iteration into multiple parallel code regions, or phases, and consider each one separately. For the NUMA nodes that we consider in this work, we define phases as regions of parallel code separated by shared-memory synchronization primitives, specifically, locks and barriers. Phases provide an opportunity for increasing energy efficiency as each phase can have different density of arithmetic to memory operations. Previous research has shown how these properties can be leveraged to throttle down the concurrency of memory-bound phases. Throttling down concurrency often reduces contention and waste due to memory-induced stall cycles in the computation [5].

We define performance in terms of the sum of the per thread instructions per cycle, over all threads executing a phase, or per phase *cumulative* IPC. Figure 1 depicts our four quad-core NUMA test system based on the AMD Opteron processor, with each processor capable of five distinct frequency levels (2, 1.7, 1.4, 1.2, and 1 GHz). We illustrate the cyclic numbering of processor sockets and cores, which is applied on AMD processor-based NUMA nodes. The system that we consider provides a rich set of thread-to-core mappings which can result in very different execution properties. For a given number of threads T used to execute a parallel program, the programmer has $\binom{16}{T}$ options to place the threads on cores. A model needs to examine all possible thread placements for all values of T up to the number of cores on the test platform, in order to derive optimal predictions of performance or other metrics. The complete configuration space where each core is considered unique, becomes prohibitively large even for a system consisting of a few multi-core processors. To address this issue we introduce a few simplifications, which we believe (a) are not a source of significant error, (b) do not leave out configurations of interest, and (c) are sufficient enough to prune the configuration space to a manageable size.

First, we assume that the four cores within a processor have identical properties, in terms of computational (pipelines, execution units) and resource (registers, caches) capacities. Hence, we only need to specify the socket that a thread is to be bound on, not the particular core inside the socket, given that each thread receives a distinct core. Second, sockets are considered identical. Therefore, we only specify the number of sockets in a configuration, subject to a specific layout which we define later in this section. Third, we only consider configurations with a bal-

anced thread placement – all multi-core processors in the system are running the same number of threads or are halted and do not participate in the computation. Imbalanced placements may introduce different degrees of contention for shared resources between groups of threads running on the same multi-core processor, thereby compromising load balancing. In practice, we find that in all cases for a given number of threads used to execute a program, a balanced thread placement is superior to an imbalanced placement.

Our model selects the number and placement of threads on cores out of a configuration space. The number of threads is used to apply *Dynamic Concurrency Throttling* (DCT) [6, 21]. DCT adjusts the number and placement of threads used in each phase of a parallel program running on a shared-memory architecture, to sustain optimal performance while reducing energy consumption. The rationale behind DCT is that certain phases with low arithmetic-to-memory operation density can benefit from running on less than the maximum number of cores, thereby reducing contention for memory bandwidth and other shared system resources. Note that DCT is an optimization that can simultaneously improve performance and lower energy consumption. In addition to thread count and thread placement, our model predicts the impact of *Dynamic Voltage and Frequency Scaling* (DVFS) on performance. DVFS [22] exploits *slack time* (core idle time during synchronization intervals) to scale the voltage and frequency of each core, to reduce energy consumption without lengthening execution time. DVFS is the predominant energy optimization methodology used in HPC and general-purpose computing systems to date [22].

Under the aforementioned assumptions the combined DCT-DVFS configuration space can be expressed as a 4-tuple, with the first three terms (processors, cores, and layout) defining the concurrency level, while the last term specifies the voltage/frequency level. The first term specifies the number of processors used for the given phase, the second defines the number of cores in each of the processors that have threads scheduled on them, and the third — the particular layout of the threads. Hence, on our test system configurations of the form (4,4,X,X) use 16 threads scheduled on the four cores of all four processors, configurations of the form (3,4,X,X) correspond to 12 threads scheduled on 4 cores of 3 processors, while configurations of the form (4,3,X,X) correspond to 12 threads scheduled on 3 cores of all four processors.

Apart from the number of cores and processors, we also identify as much as four distinct thread layouts per given processor-core combination. Figure 2 shows the layouts that we consider, assuming program execution with 16 threads. In configurations of the form (4,4,4,X) the first four threads are scheduled on the four cores of socket 0, the next four threads on socket 1, the next four threads on socket 2 and the final four threads on socket 3. In other words, the thread mapping follows the numbering of cores in the system, so that each thread with a given ID maps to a core with the same ID. The layout term (third term in the 4-tuple) specifies the granularity of interleaving of threads to sockets in the system. A layout of '1' implies a one-way interleaving, meaning an interleaving of threads across sockets at a granularity of a single thread. A layout of '2' defines a two-way interleaving granularity, while a layout of '4' defines a four-way interleaving granularity, where each socket is filled completely before threads are scheduled on the subsequent socket.

Layout '3' produces a hybrid distribution where the last four threads need to be interleaved at one-way granularity while the rest of the threads are interleaved at three-way granularity. As a result, this layout that we discuss in the next section. The four layouts depicted in Figure 2 generalize to configurations of less than 16

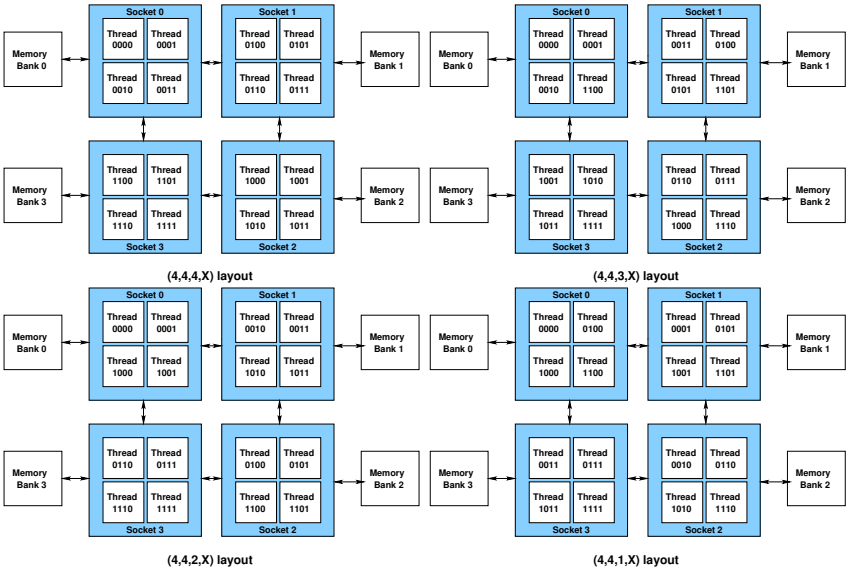

Figure 2. Layouts of 16-thread executions considered in our model.

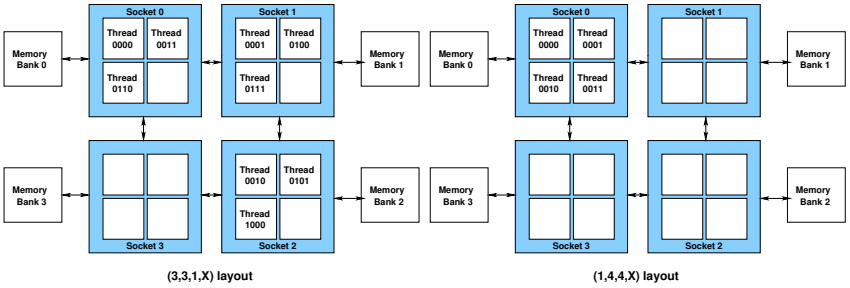

Figure 3. Examples of layouts with less than 16 threads.

threads. Figure 3 shows an example of a 9-thread layout and a 4-thread layout. Additionally, configurations that only utilize a single processor have a single layout, as all threads need to be scheduled on the same processor. Similarly configurations with only one core per processor, e.g. (4,1,1,X), are also limited to a single layout, namely one-way interleaving.

Finally, the last term in the tuple identifies the frequency level of all cores in the configuration. Although current processors support independently manipulating the voltage/frequency level of each core within a multi-processor as well as between processors, we only consider setting voltage and frequency levels globally, across all cores. Our reasoning is that parallel HPC applications attempt to distribute their workload equally between threads in a parallel phase. As such, there is no motivation to attempt to execute threads in a given phase on cores with different frequency levels, while these threads perform computation and this computation is balanced by the runtime system or application-specific code. More subtly, there can be undesirable hardware effects from executing a phase on processors which are set to different frequency levels. For example, on our test system the on-chip memory controller is clocked down depending on the lowest frequency set among the four multi-core processors on the system, .

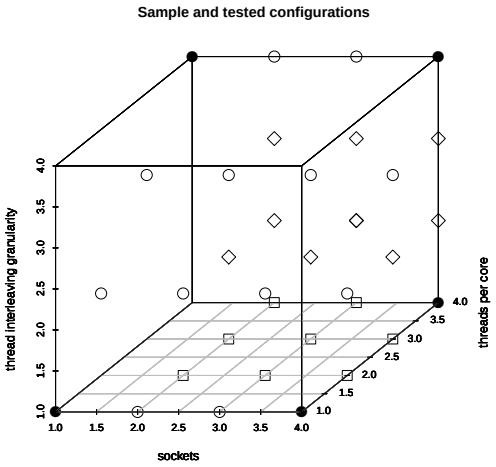

Figure 4. Sampled configuration space with samples at maximum and minimum interleaving granularity. Sampled configurations shown as black circles, correspond to the 3-tuples (1,4,4), (4,4,4), (1,1,1), (4,1,1), and (4,4,1). Predicted configurations are shown as white circles (first sampling step), white squares (second sampling step), and white diamonds (third sampling step).

3. Performance Model

We derive a model that predicts performance in response to varying the number of threads used to execute application phases via dynamic concurrency throttling, the layout of threads on cores, and the voltage and frequency of cores. We discuss the details of the model in the following subsections.

3.1 DCT Model

We construct our DCT performance model around the assumption that varying one term in the configuration tuple, while keeping all other terms constant, produces a linear change in *per thread* IPC ($ptIPC$).

(1)

. For the following discussion we omit the DVFS term in the tuple and assume it takes on the maximum value (2 GHz on our test platform). First, we sample per thread IPC at the end points in the configuration space when the frequency is maximized, the number of sockets used is either minimum or maximum, and the layout uses the maximum interleaving granularity, that is, the layout clusters as many threads as possible on each socket. The selected configurations are (4,4,4), (4,4,1), (1,4,4), (4,1,1), and (1,1,1). Figure 4 shows the relevant sampled configurations with black circles and as well as the relevant IPC estimations.

The samples are used to linearly model the per thread IPC values along the edges of the section of the configuration space depicted in Figure 4. The produced intermediate values are then used as end points for the linear estimation of the inner values. It does not matter which intermediate values are used as end points as both calculations produce the same result. For the next step we sample per thread IPC of the configuration with maximum number of sockets and cores and the layout that implements the finest possible interleaving of threads between sockets, namely (4,4,1), which corresponds to one-way interleaving. To estimate IPC for all

configurations with the one-way interleaving layout, we perform the same procedure as before, but replace the sampled IPC at (4,4,4) with the one that we obtain from configuration (4,4,1). We can reuse the other samples as those configurations have a single layout, in which case the maximum-way interleaving and the minimum-way interleaving layouts coincide. Figure 4 plots the relevant tested section of the configuration space with white squares.

Finally, some processor-core combinations can have more than two layouts. The previous step of the modeling process either sampled or estimated the IPC of all configurations with maximum (four-way) and minimum (one-way) interleaving, therefore we can use these values to linearly estimate IPC for configurations with other interleavings (two-way and three-way). Figure 4 plots the relevant tested configurations with white diamonds.

3.2 DVFS Model

For the DVFS model we assume that varying frequency produces the same *relative* change in IPC regardless of the other three terms in the tuple representing a configuration. We leverage this assumption by empirically selecting one of the 34 DCT configurations discussed above and sampling the per phase IPC produced at a lower frequency (in our experiments we use configuration (4,1,1,1.4)). We record the per phase IPC scaling factor and linearly estimate the scaling factors for the other frequency levels. We then apply the resulting scaling factors to the IPC estimations for all configurations. An important observation is that the IPC at different frequency levels cannot be directly compared, as by definition the length of a processor's cycle changes with the frequency, whereas memory access rates per instruction and memory latency remain unchanged. To address this issue we normalize all IPC measurements to the cycle length measured at the maximum frequency of the processor (2 GHz, on our test platform). As discussed above, IPC samples are taken for each phase in the computation. Thus, we can identify properties of each phase separately and distinguish between phases with different density of arithmetic to memory operations.

The accuracy of the proposed model can be adversely affected if different DCT configurations of a phase exhibit varying contention for resources, such as memory bandwidth. In such a case, the IPC of configurations with many threads can change in a drastically different way in response to changes in the DVFS level, compared to configurations with a few threads. If memory access density or thread contention for resources in suspected to be significant, the model can easily be extended to sample more configurations. Here, as with most other models, there is a trade-off between overhead and accuracy. In our experiments we did not find memory access density and contention to affect our model's prediction accuracy. A single sample from a configuration with four threads per socket produces satisfactory accuracy for all configurations. Another trade-off is selecting the frequency at which to sample IPC. We find that sampling at a lower frequency produces slightly better accuracy, as taking samples at frequencies that are farther apart can provide a better estimate of the overall trend. However, as these samples can be taken online, reducing the frequency too much will result in unnecessary increase in overhead.

3.3 Model Discussion

While our definition of performance is *cumulative* IPC, we sample and model *per thread* IPC. We convert per thread to cumulative IPC depending on the number of threads in the given configuration.

To motivate the linear change in IPC when moving between layouts with one-way, two-way and four-way interleaving, we observe that these layouts tend to produce a linear change in traffic over the HyperTransport links. For example, interleaving threads at two-way granularity can produce as much as double the traffic over the HyperTransport links, when compared to four-way interleaving, while one-way interleaving can quadruple that traffic. From an application's point of view, the increased traffic results in increased stalls due to accessing of remote memory, which in turn have a negative effect on performance. It is important to note that the rate of change of traffic and performance are not identical. For example, a doubling of the traffic over the HyperTransport links can result in a 10% increase in execution time.

The HyperTransport traffic generated by the threads with one-way interleaving have a dominating effect and as a result in our experiments per thread IPC is usually between that of one-way and two-way interleaving layouts. that a good estimate for IPC of layouts with three-way interleaving is 20% below the IPC of layouts with two-way interleaving.

4. Experimental Analysis

We evaluate our prediction model as it applies to a real NUMA system executing HPC workloads. We begin with a brief overview of our implementation and experimental setup. Next, we analyze the properties of the configuration layouts that we consider. Finally, we evaluate the performance of our model and provide a head-to-head comparison to a state-of-the-art model proposed by Curtis-Maury, *et al.* [5].

4.1 Experimental Setup

We have implemented our model on a real system, using only portable components. Our experimental platform has four AMD Opteron 8350HE quad-core processors, for a total of sixteen cores. Each core has a private L1 and L2 cache of size 128 KB (64 KB data + 64 KB instruction) and 512 KB respectively. Each processor has a 2 MB L3 cache that is shared among its four cores. Each core has five frequency levels between 2 and 1 GHz. The system has 64 GB of memory distributed evenly between its four nodes and maintains coherence between the 16 GB memory banks of each node using the on-chip memory controllers and HyperTransport links between the nodes. The system runs Linux kernel version 2.6.31, which has integrated support for measuring hardware event rates, such as IPC. Full system energy is collected per run using a WattsUp Pro power meter. In all experiments, power data is collected by an auxiliary machine, thus eliminating any performance loss on the test system.

In our experiments we use benchmarks representative of the HPC domain. Specifically, we use seven benchmarks of the OpenMP version of the NAS Parallel Benchmark suite (3.3). All benchmarks are compiled at class size C, except for IS, which is compiled at class size D. We use PAPI [23] in conjunction with TAU [24] to facilitate recording of hardware event rates. Linking with the TAU libraries hides all the complexities of instrumenting the NAS benchmarks behind a few simple calls, while incurring negligible overhead. We use the Linux processor affinity system call, *sched_setaffinity()*, and *omp_set_num_threads()* to manage concurrency. Finally, the voltage-frequency level is set using the *cpufrequtils* library.

Our performance model relies on hardware event samples collected at the granularity of phases. Previous research has shown how to exploit the iterative nature

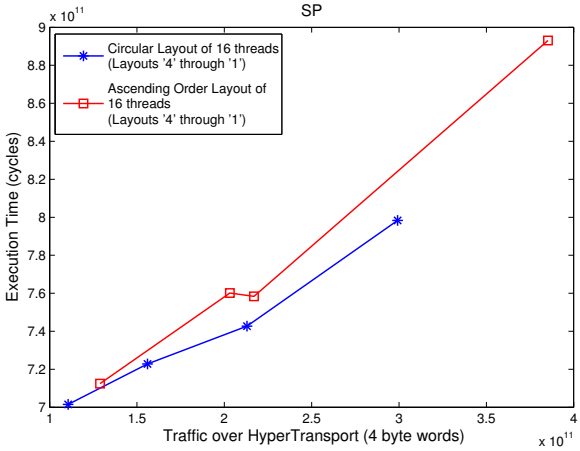

Figure 5. Linear relationship between HyperTransport traffic and execution time of SP benchmark

of HPC codes to collect hardware event rates from the first few iterations of each phase executed using different configurations [5]. The samples collected online are used as input to a performance model, which determines the best per phase configuration and applies it to the remaining iterations of the code. However, this approach cannot be readily applied if the workload is executed on a NUMA system Running any phase in a configuration different from the one used to initialize the data can result in severe performance penalties due to remote data accesses. Thus, sampling a phase running is such a configuration will result in a radically different result compared to the case when initialization and the first iteration of the main loop are executed with the same configuration.

To address this issue, To collect the six samples needed for our model we run six instances of the truncated benchmark, each running the parallel initialization and a single iteration of the main loop using the same configuration. This technique can be readily applied to any iterative application. In order to ensure fair comparison we apply the same approach to the

In an effort to further decrease sampling overhead of our model, we estimate, rather than directly sample, per thread IPC at the bottom of the spectrum. We choose to sample configuration (3,1,1,2), which along with configuration (4,1,1,2) can be used to estimate the IPC of configuration (1,1,1,2). Model generation remains otherwise as described in the previous section. The optimization usually results in a three-fold decrease in overhead for that sample, as most benchmarks see a linear or superlinear speedup at the bottom end of the configuration space.

4.2 Analysis of Layouts

In Section 2 we discussed the layouts that we consider, based on different thread interleaving between processor sockets. In this section we show the significance of thread layouts for performance. Specifically, we demonstrate that layouts that a) map threads in a circular manner that conforms to the HyperTransport link layout and the numbering of sockets on our test system and b) use coarse-grain interleaving, achieve significant reduction in traffic over the interconnect. Figure 5 relates traffic and execution time for one of the benchmarks – SP – where layouts produce a significant change in performance. We plot the performance of the circular layouts that we consider in our model, using different degrees of interleaving and alternative layouts that ignore socket numbering. The latter interchange sockets 2 and 3 (Figure 1) while mapping threads in ascending order (labeled 'Ascending' on

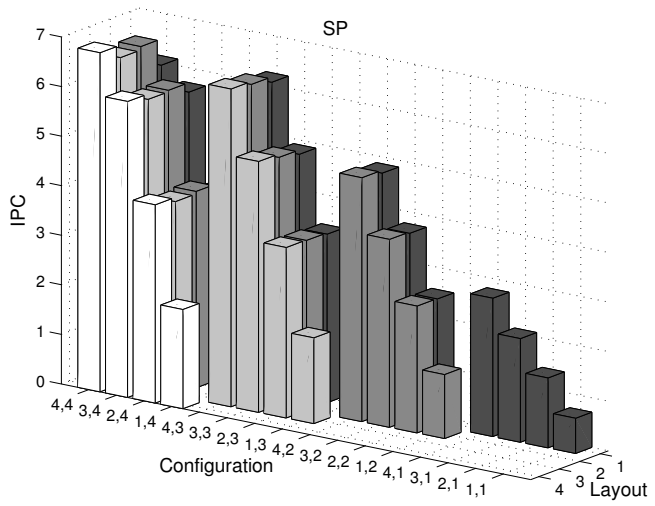

Figure 6. IPC of SP benchmark

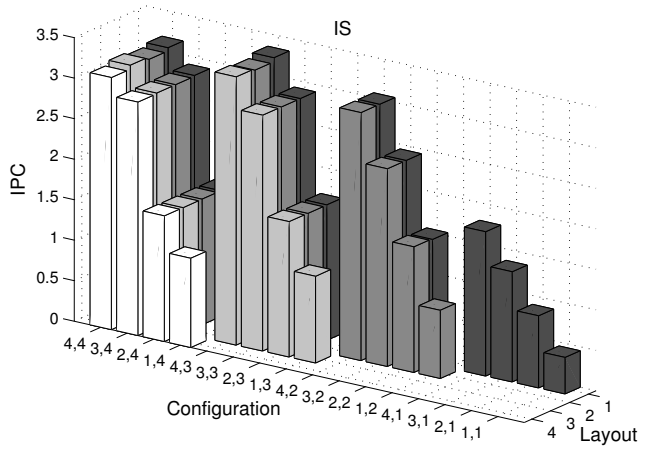

Figure 7. IPC of IS benchmark

Figure 5). We observe a linear relationship between execution time and traffic and a linear relationship (with one exception, three-way interleaving in the 'Ascending' layout) between execution time and degree of interleaving. The 'Ascending' layouts that are ignorant of HyperTransport link and socket layout, produce significantly more traffic and hence longer execution, especially with fine-grain one-way interleaving.

It is important to note that not all benchmarks see an increase in HyperTransport traffic and worse performance with fine-grain interleaving layouts. In IS, for example, traffic over the interconnect remains largely constant between the four circular layouts. As shown in Figure 7 the layout with one-way interleaving has a slightly higher IPC, due to better utilization of bandwidth to local memory within each quad-core processor. In this case, as remote memory accesses are an insignificant performance factor, the 'Ascending' order layouts have only marginally worse performance than the 'Circular' ones.

4.3 Application Scalability Analysis

Before evaluating the prediction accuracy and associated benefits of our model, we briefly analyze the scalability of the applications on our testbed. Due to space

Table 1. Speedup Results

Config.	BT	CG	FT	IS	LU	MG	SP
(4,4,4,2)	**13.94**	**15.86**	**11.64**	6.84	11.48	**10.01**	**9.14**
(4,4,1,2)	12.71	15.62	11.57	6.95	**11.61**	9.58	8.10
(4,3,3,2)	11.58	13.36	9.97	**7.27**	9.41	9.32	8.79
(3,4,4,2)	11.39	13.35	8.99	6.28	8.54	8.79	8.12
(4,1,1,2)	4.21	4.57	3.99	3.93	3.84	3.86	3.92
(1,4,4,2)	3.82	4.96	3.50	2.43	3.16	3.15	2.82

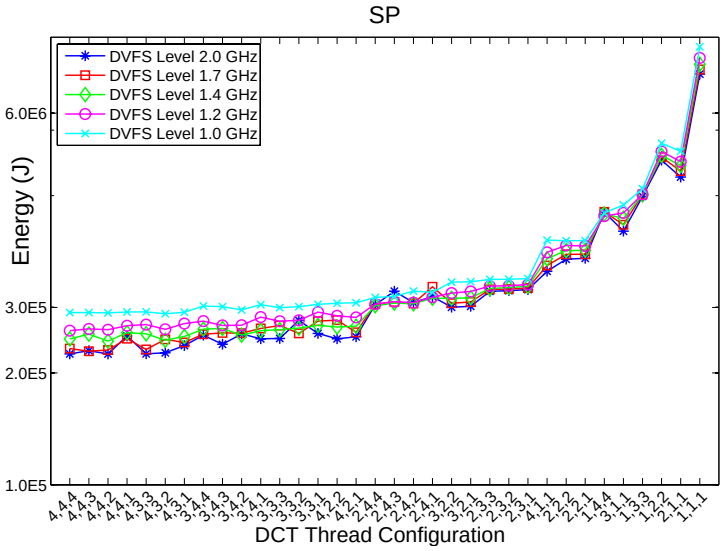

Figure 8. Energy efficiency of SP

limitations we cannot display data for all of the hundred and seventy (34 DCT configurations with 5 DVFS levels each) configurations.

Table 1 presents the speedup for a few relevant configurations for the seven benchmarks All benchmarks scale reasonably well with some (BT, CG) scaling close to linearly all the way to 16 threads. All benchmarks scale linearly or superlinearly up to 4 threads, at which point, scalability starts to level off for some benchmarks. As can be seen from the last four rows in the table most benchmarks perform better when threads are spread around the system, rather than grouped on as few multi-processors as possible. This is especially evident in configurations with 4 threads. We attribute this to the higher contention in the memory hierarchy that grouping on the same processor produces. BT, MG, and SP see a significant performance difference between layouts.

Figure 6 and Figure 7 present scalability results of the entire DCT space for SP and IS, respectively. Most benchmarks exhibit behavior similar to SP, with generally uniform scalability and moderate sensitivity to thread placement. IS, in contrast, shows poor scalability and high sensitivity to thread placement. As a result, IS is the only benchmark that sees its maximum performance at a configuration with less than 16 threads. In general, the relative performance of layouts for all benchmarks remains the same between configurations with a different number of threads each. For example, in SP both (4,4,4,2) and (4,3,3,2) have higher IPC than the respective configurations with one-way and two-way interleaving layouts.

For all benchmarks the most energy-efficient configuration coincides with the most performance-efficient. We attributed this to (1) the good scalability of all benchmarks and (2) the relatively high idle power of our test system. For example, the idle power of our test machine is 345 Watts, power usage for SP running on configuration (1,1,1,2) consumes 460 Watts, while running on configuration (4,4,4,2) - 591 Watts. Thus, any power savings from executing at less than optimal

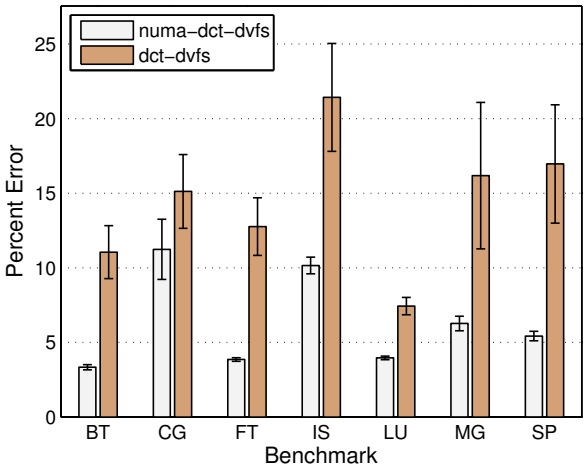

Figure 9. Weighted average prediction error

concurrency is insufficient to compensate for the additional time that idle-power is incurred. Similarly, as lower DVFS levels do not produce better performance, they consume more energy than the highest DVFS level (Figure 8).

4.4 Performance Prediction Evaluation

We provide a head-to-head comparison of our model to DPAPP, a state-of-the-art DCT-DVFS prediction model developed by Curtis-Maury et al. [5]. Both our model and DPAPP predict performance in response to varying the number of threads, the layout of threads on cores, and voltage-frequency during application execution. Furthermore, both models use linear regression. Unlike our model, DPAPP predicts performance by sampling selected hardware event rates that are strong predictors of performance in a statistical sense. These hardware events need to selected separately for each target hardware platform, through rigorous statistical analysis. Our model uses solely IPC measurements, a universally available metric, therefore it is superior in portability and applicability to DPAPP.

We evaluate first the overall accuracy of both models. We train our NUMA-aware DCT-DVFS model, as outlined in Section 3, by taking six samples from truncated versions of the seven benchmarks in our test suite. For DPAPP, following the process outlined in [5], we first execute a truncated version of the UA benchmark in all hundred and seventy configurations in order to estimate the coefficients needed for the multivariate regression model. For each benchmark we then take three samples from truncated versions running at configurations (4,4,4,2), (1,4,4,2), and (4,1,1,1.7).

Figure 9 presents the overall error between predicted and actual cumulative IPC of a production run of the benchmarks for our model (labeled 'numa-dct-dvfs') and DPAPP (labeled 'dct-dvfs'). As both models can make per phase predictions, we show weighted average error where longer running phases have a higher contribution to the average. In all but CG, our model is significantly more accurate, with an average error of less than 7%. On three of the seven benchmarks error is less than 4% with negligible standard deviation. In contrast, in all benchmarks but LU DPAPP sees considerable error coupled with high standard deviation. In extreme cases, in the MG and SP benchmarks, the predicted IPC for a given configuration is more than 200% away from the actual value.

Both models see a high error with the IS benchmark, although it is a factor

of two higher for DPAPP. We attribute this to the complex scalability properties that the benchmark exhibits. As Figure 7 demonstrates, IS is extremely sensitive to thread placement. For example, two configuration with six threads (3,2,1,2) and (2,3,1,2) are a factor of two away in terms of performance. Additionally, (3,2,1,2) outperforms a configuration with twice as many threads – (3,4,1,2). CG, on the other hand, poses a very different prediction challenge. It scales almost linearly to 16 cores, however, scalability changes from superlinear to linear along some dimensions of the space, most significantly in configurations with 8 and 12 threads. Our model is unable to capture the subtle hardware-software interaction that allows such effects and as a result this is where our model sees the highest error – around 11% on average. As discussed below, this turns out to be of no practical consequence, as both models can correctly identify that the best performing configuration is at maximum concurrency.

4.5 Model Application

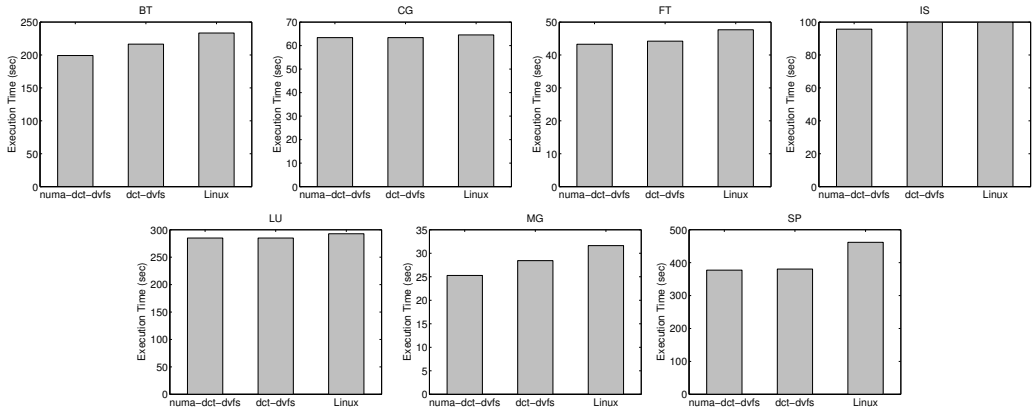

Figure 10. Execution time of best configuration

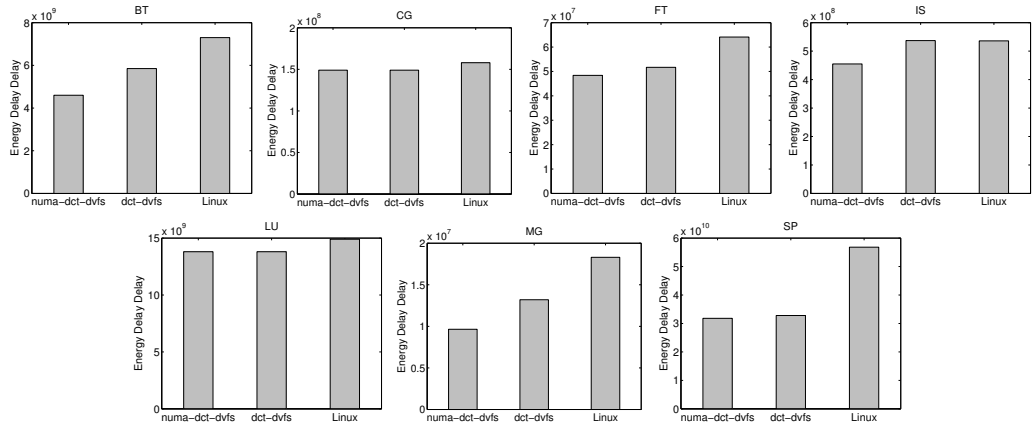

Figure 11. Energy delay square of best configuration

We explore the benefits of applying our prediction model to maximize performance and energy efficiency. We evaluate our model versus DPAPP and compare the performance of the two models to the one achieved by the Linux scheduler. For the base case we set the OpenMP concurrency to maximum and eliminate any restrictions to the scheduling of threads to cores. We couple the default scheduling

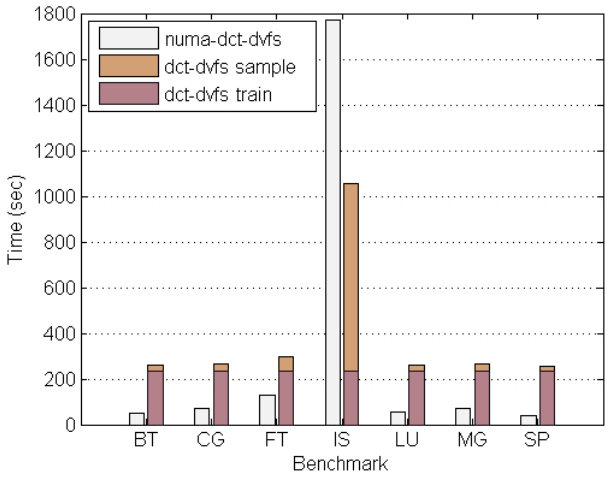

Figure 12. Model training time

policy with the in-kernel *Ondemand* power controller, that dynamically controls voltage-frequency levels based on CPU load [25]. We use the per phase predictions of our model and DPAPP to identify the configuration that minimizes overall execution time.

Figure 10 shows the execution time of the best configuration identified by each model, as well as that produced by the Linux scheduler. Our model identifies the optimal configuration in all experiments, whereas in all but two of the seven cases DPAPP chooses a suboptimal configuration. The DPAPP model still chooses a configuration that is usually not too far off the optimal (4% slower than optimal on average). The poor performance of the Linux scheduler is due to its use of 'ascending order' layouts to schedule threads to cores –thread layouts that ignore the actual layout and numbering of processor sockets– and as a result performance suffers in benchmarks where traffic over HyperTransport is significant.

As most benchmarks scale reasonably well all the way to 16 threads, the relative change in IPC along most dimensions of the configuration space is quite large. As a result even with its substantially larger error, DPAPP is usually able to correctly predict the number of threads in the optimal configuration. However, its error is usually prohibitively large to correctly predict performance between different layouts of the same configuration or even between performance of configurations with the same number of threads, such as (4,3,X,X) and (3,4,X,X). On the IS benchmark DPAPP fails to identify the optimal number of threads and produces worse performance than the default Linux behavior.

Figure 11 shows the ED^2 produced by the configuration selected by each model. ED^2 represents the product of energy consumption with the square of execution time, which is a popular metric for energy-efficiency in high performance computing. Our model achieves ED^2 savings of around 40% in three of the seven benchmarks and above 25% on average combined with an average performance improvement of over 10%. In contrast, DPAPP achieves an ED^2 savings of 17% coupled with a performance increase of 6% on average when compared to Linux.

One of the most important advantages of our model is that it does not incur a long training overhead. Figure 12 shows the sampling time of our model as well as the sampling and amortized training overhead of DPAPP. The anomaly in the IS benchmark is due to the fact that a significant computationally intensive portion of the application is not a part of the main iteration loop and is not reported in the official results of the NAS suite. Thus, even executing a truncated version of

that benchmark results in significant computation time. This behavior is atypical in the applications that we are targeting, therefore we exclude it from the following discussion.

The training overhead required to derive the coefficients for the DPAPP multivariate linear model, even when amortized over seven benchmarks, is quite significant – it accounts for close to 90% of total model overhead. Our model, in contrast, does not incur training overhead. It produces an estimation directly from the six samples taken from points in the configuration space. As a result, its overhead is 75% lower on average compared to that of DPAPP. As another point of comparison, the derivation time of our model is less than 3% of the overhead incurred by a sequential search through the entire configuration space.

5. Related Work

Lee and Brooks [7, 8] use spline regression models for microarchitectural design space exploration. Their techniques, similarly to ours, rely on sampling the design space and performing statistical inference analysis. The regression model used in their work, similarly to ours, enables efficient performance prediction. The domain of this work however requires typically offline training with thousands of samples, whereas our domain requires few samples and can be applied with little overhead for online optimization. We further consider optimization criteria besides architectural configuration parameters, including thread assignment to cores and data layouts, as well as criteria besides performance, including power and energy-delay. We also constrain our prediction models to use portable metrics that can be sampled on a wide variety of processors.

Li, *et al.*, [26] and Ipek, *et al.*, [27] use artificial neural networks (ANNs) as black-box models for microarchitectural design space exploration. Li, *et al.* [26], similarly to our work, use ANNs as models for online performance prediction, that they apply to task partitioning and scheduling for HPC clusters. Their work does not consider NUMA architectures, core and data layout effects, or power metrics. ANNs tend to simplify the process of constructing performance models, whereas regression-based models tend to be more computationally efficient for online program adaptation. The trade-off's between ANNs and regression models as performance and power predictors remain unclear. Work by Singh *et al.* [28], indicates comparable prediction accuracy and performance between the two approaches, in the context of prediction for power-performance adaptation.

Barnes, *et al.*, [4] use linear regression for predicting the scalability of parallel applications on large processor counts from a few sample executions. Their models, similarly to ours, rely on very few training samples, typically only three. Their work considers only performance prediction in response to varying the number of processors and using one parameter (execution time, per processor information, or critical path length), whereas our model considers prediction for multiple target optimization criteria and along multiple dimensions of adaptation, while still using one parameter (per thread IPC). Yang, *et al.*, [12] propose a similar model for parallel applications, focusing on cross-platform performance prediction. Their work leverages partial application execution, which we also use in our model for faster sampling of configurations used for prediction, albeit in a different context (NUMA multi-core systems).

Curtis-Maury, *et al.* [5, 6, 21] use linear regression models for online power-performance adaptation of multithreaded codes on multi-core architectures. Our work differs from their research in several aspects. First, we introduce an additional dimension in the configuration space by differentiating between thread layouts. Ad-

dressing the performance implications of the layout of threads on cores is especially important in the context of a NUMA system, where data and thread layout can have a significant impact on performance and energy-efficiency. Second, our prediction model relies on a universally applicable metric, which eliminates the need to identify architecture-specific hardware event rates as dominant performance factors. Finally, unlike other techniques, no lengthy training stage or exhaustive search of the configuration space is required, drastically reducing overhead.

6. Conclusion

In this paper we have presented a scalable statistical "black-box" model for predicting the performance of parallel programs on multi-core NUMA systems, in response to varying several tunable execution-time parameters –number and layout of threads on cores, and core voltage-frequency. We have demonstrated the merits of applying the model to accurately predict the behavior of parallel applications. We show that using our framework on a physical 16-core NUMA system results in significant energy and ED^2 savings (9% and 25% respectively) along with performance improvement of 10% on average. Our prediction model proves substantially more accurate than previous efforts, in part by identifying the layout of threads as a significant performance factor. At the same time, our model is drastically easier to apply and port across systems. Our future work involves further investigation of NUMA-related performance issues, in particular with respect to alternative data placement policies. We plan to further explore the optimization space, focusing in particular on discovering per phase dynamic optimal configurations, subject to layout constraints, as well as cross-input performance prediction.

References

[1] V.S. Adve and M.K. Vernon, *Parallel Program Performance Prediction using Deterministic Task Graph Analysis*, ACM Trans. Comput. Syst. 22 (2004), pp. 94–136.

[2] D.J. Kerbyson, H.J. Alme, A. Hoisie, F. Petrini, H.J. Wasserman, and M. Gittings, *Predictive Performance and Scalability Modeling of a Large-Scale Application*, in *SC'01: Proc. of the 2001 ACM/IEEE Conference for High Performance Computing, Networking, and Storage*, 2001.

[3] G. Marin and J. Mellor-Crummey, *Cross-Architecture Performance Predictions for Scientific Applications using Parameterized Models*, in *SIGMETRICS'04/Performance'04: Proc. of the Joint International Conference on Measurement and Modeling of Computer Systems*, 2004.

[4] B.J. Barnes, B. Rountree, D.K. Lowenthal, J. Reeves, B. Supinskide , and M. Schulz, *A Regression-Based Approach to Scalability Prediction*, in *ICS'08: Proc. of the 22nd Annual International Conference on Supercomputing*, 2008.

[5] M. Curtis-Maury, A. Shah, F. Blagojevic, D.S. Nikolopoulos, B.R. Supinskide , and M. Schulz, *Prediction Models for Multi-Dimensional Power-Performance Optimization on Many Cores*, in *PACT'08: Proc. of the 17th International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[6] M. Curtis-Maury, F. Blagojevic, C.D. Antonopoulos, and D.S. Nikolopoulos, *Prediction-Based Power-Performance Adaptation of Multithreaded Scientific Codes*, IEEE Transactions on Parallel and Distributed Systems 19 (2008), pp. 1396–1410.

[7] B.C. Lee and D.M. Brooks, *Accurate and Efficient Regression Modeling for Microarchitectural Performance and Power Prediction*, in *ASPLOS-XII: Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct., 2006.

[8] B.C. Lee and D.M. Brooks, *Illustrative Design Space Studies with Microarchitectural Regression Models*, in *HPCA-13: Proc. of the 13th International Conference on High Performance Computer Architecture*, 2007.

[9] B.C. Lee, D.M. Brooks, B.R. Supinskide , M. Schulz, K. Singh, and S.A. McKee, *Methods of Inference and Learning for Performance Modeling of Parallel Applications*, in *PPOPP'07: Proc. of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2007.

[10] Z. Wang and M.F. O'Boyle, *Mapping Parallelism to Multi-Cores: A Machine Learning Based Approach*, in *PPOPP'09: Proc. of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2009.

[11] J. Zhai, W. Chen, and W. Zheng, *PHANTOM: Predicting Performance of Parallel Applications on Large-Scale Parallel Machines using a Single Node*, in *PPoPP '10: Proc. of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2010.

[12] L.T. Yang, X. Ma, and F. Mueller, *Cross-Platform Performance Prediction of Parallel Applications Using Partial Execution*, in *SC'05: Proc. of the 2005 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2005.

[13] S.R. Alam, R.F. Barrett, M.R. Fahey, J.A. Kuehn, O. Bronson Messer, R.T. Mills, P.C. Roth, J.S. Vetter, and P.H. Worley, *An Evaluation of the Oak Ridge National Laboratory Cray XT3*, International Journal of High Performance Computing Applications 22 (2008), pp. 52–80.

[14] S.R. Alam, J.A. Kuehn, R.F. Barrett, J.M. Larkin, M.R. Fahey, R. Sankaran, and P.H. Worley, *Cray XT4: An Early Evaluation for Petascale Scientific Simulation*, in *SC'07: Proc. of the 2007 ACM/IEEE Conference on High Performance Computing, Networking, Storage, and Analysis*, 2007.

[15] A. Baumann, P. Barham, P.E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, *The Multikernel: A New OS Architecture for Scalable Multicore Systems*, in *SOSP'09: Proc. of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, 2009.

[16] S. Saini, A. Naraikin, R. Biswas, D. Barkai, and T. Sandstrom, *Early Performance Evaluation of a Nehalem Cluster using Scientific and Engineering Applications*, in *SC'09: Proc. of the ACM/IEEE Conference on High Performance Computing, Networking, Storage and Analysis*, 2009.

[17] K.J. Barker, K. Davis, A. Hoisie, D.J. Kerbyson, M. Lang, S. Pakin, and J.C. Sancho, *A Performance Evaluation of the Nehalem Quad-Core Processor for Scientific Computing*, Parallel Processing Letters 18 (2008), pp. 453–469.

[18] D. Hackenberg, D. Molka, and W.E. Nagel, *Comparing cache architectures and coherency protocols on x86-64 multicore SMP systems*, in *MICRO 42: Proc. of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009.

[19] C. McCurdy and J. Vetter, *Memphis: Finding and Fixing NUMA-Related Performance Problems on Multi-core Platforms*, in *ISPASS'2010: Proc. of the IEEE International Symposium on Performance Analysis of Systems and Software*, 2010.

[20] S. Siddha, V. Pallipadi, and A. Mallick, *Process scheduling challenges in the era of multi-core processors*, Intel Technology Journal 11 (2007).

[21] M. Curtis-Maury, J. Dzierwa, C. Antonopoulos, and D. Nikolopoulos, *Online Power-Performance Adaptation of Multithreaded Programs using Hardware Event-Based Prediction*, in *ICS'06: Proc. of the 20th ACM International Conference on Supercomputing*, Jun., 2006.

[22] R. Springer, D.K. Lowenthal, B. Rountree, and V.W. Freeh, *Minimizing Execution Time in MPI Programs on an Energy-Constrained, Power-Scalable Cluster*, in *PPOPP'06: Proc. of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Mar., 2006.

[23] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci, *A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters*, in *SC'2000: Proc. of the 2000 ACM/IEEE International Conference for High Performance Computing, Networking, and Storage*, Nov., 2000.

[24] S.S. Shende and A.D. Malony, *The Tau Parallel Performance System*, Int. J. High Perform. Comput. Appl. 20 (2006), pp. 287–311.

[25] V. Pallipadi and A. Starikovskiy, *The Ondemand Governor*, in *Proc. of the Ottawa Linux Symposium*, Jul., 2006.

[26] J. Li, X. Ma, K. Singh, M. Schulz, B.R. Supinskide , and S.A. McKee, *Machine Learning Based Online Performance Prediction for Runtime Parallelization and Task Scheduling*, in *ISPASS'09: Proc. of the IEEE International Symposium on Performance Analysis of Systems and Software*, 2009.

[27] E. Ipek, S.A. McKee, K. Singh, R. Caruana, B.R.d. Supinski, and M. Schulz, *Efficient Architectural Design Space Exploration via Predictive Modeling*, ACM Trans. Archit. Code Optim. 4 (2008), pp. 1–34.

[28] K. Singh, M. Curtis-Maury, S.A. McKee, F. Blagojevic, D.S. Nikolopoulos, B.R. Supinskide , and M. Schulz, *Comparing Scalability Prediction Strategies on an SMP of CMPs*, in *EURO-PAR 2010 Parallel Processing Conference*, 2010.